PrUcess (Processing unit through UART)

Introduction

Pr**U**cess is a **processing** unit that executes commands (unsigned arithmetic operations, logical operations, register file read & write operations) which are received from an external source through **UART** receiver module and it transmits the commands' results through the **UART** transmitter module.

This is a full ASIC design project (from RTL to GDS). It goes through the ASIC design flow from frontend to backend:

1. System's architecture design.
2. Synthesizable Verilog RTL modelling (behavioral modelling, structural modelling, and FSM coding) of all the system blocks from scratch (UART transmitter and receiver, integer clock divider, ALU, register file, parametrized data and bit synchronizers for solving CDC issues, reset synchronizer, and system's main controller).
3. Solving CDC issues by using synchronizers.
4. Functional verification using self-checking testbenches and automated Python verification environments and running the testbenches using Modelsim.
5. Logic synthesis using Synopsys Design Compiler & 130 nm technology node.
6. Formal verification post logic synthesis using Synopsys Formality.
7. Design for testability (DFT) using Synopsys DFT Compiler.
8. Formal verification post DFT using Synopsys Formality.
9. Physical design (floor planning, power planning, placement, CTS, routing, timing closure, and chip finishing) using Cadence innovus.
10. Formal verification post physical design using Synopsys Formality.

Table of Contents

System's Specifications

UART is a standard serial communication protocol widely used in many applications. Oversampling is a technique used in UART receivers to improve the accuracy and reliability of the received data. In a UART receiver, data is received as a series of binary bits that are transmitted asynchronously with respect to a clock signal. To correctly interpret the received data, the receiver must sample the incoming signal at the correct time to capture the correct value of each bit. Oversampling involves sampling the incoming signal at a higher frequency than the baud rate of the transmitted data. This means that multiple samples are taken during the transmission of each bit, allowing the receiver to more accurately determine the timing and value of each bit. Oversampling also helps to mitigate the effects of noise and other signal distortions that can cause errors in the received data. By taking multiple samples of each bit, the receiver can detect and correct for these errors, improving the overall reliability of the data transmission.

The system includes two asynchronous clock domains (reference clock domain and UART clock domain). The command is received by the UART receiver, then it is sent to the system controller through a synchronizer (to solve the CDC issues) to decode and execute the command and then it sends the result to the UART transmitter through a synchronizer which will finally transmit it serially.

System's Parameters

Parameter	Default Value	Description
DATA_WIDTH	8	It is the size of: registers, ALU operands, UART transmitter frames, and UART receiver frames.
REGISTER_FILE_DEPTH	16	The number of registers in the register file.
SYNCHRONIZER_STAGE_COUNT	2	The number of stages in a synchronizer.
oversampling_prescale	the default value after resetting the system is 8	The ratio between the frequency of the UART receiver clock and the frequency of the UART transmitter clock.

System's Clock Domains

Clock Domain	Clock Names	Modules	Frequency
Reference clock domain	• reference_clk • ALU_clk	• System controller • ALU • Register file	Reference clock frequency = ALU clock frequency = 40 MHz
UART clock domain	• UART_clk • UART_transmitter_clk	• UART transmitter • UART receiver • Clock divider	• UART transmitter clock frequency = 115.2 KHz (standard baud rate) • UART clock frequency = oversampling_prescale * 115.2 KHz = 32 * 115.2 KHz = 3.6864 MHz

Note that the oversampling prescale can have the values (8, 16, or 32) but 32 is used in the simulations and backend flow to ensure that the UART receiver is functioning correctly in the worst case (highest clock frequency).

System's Components:

1. UART: It consists of UART receiver which receives the commands and UART transmitter that transmits the commands' results.
2. Clock divider: An integer clock divider which can divide the source clock up to division ratio of 32. It is used to divide the UART clock to produce UART transmitter clock with division ratio equal oversampling prescale.
3. ALU: It executes unsigend arithmetic operations and logical operations.
4. Clock gating cell: It is used to gate the ALU clock because there is significant time in which the ALU is not in operation (because the ALU operates on a very fast clock compared with the UART, so it waits long time to receive a new command).
5. Register file.
6. System controller: It is the main controller of the system. It consists of UART transmitter controller and UART receiver controller. The UART transmitter controller controls the UART transmitter by sending to it the data to be sent serially after it is ready (ALU result or register file data). The UART receiver controller controls the ALU and register file control signals based on the received frames from the UART receiver.
7. Reset synchronizer: It is used to synchronize the global reset to all clock domains.
8. Bus synchronizer: This module can be used to synchronize a single bit or a grey encoded bus between two asynchronous clock domains. It is a generic module (setting BUS_WIDTH = 1, means that it is a single bit synchronizer).
9. Data synchronizer: It is used to synchronize a bus by using a bit synchronizer and pulse generator to synchronize the bus's data valid signal.

ALU Operations:

1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)
5. Bit-wise AND (&)
6. Bit-wise OR (|)
7. Bit-wise NAND (~&)
8. Bit-wise NOR (~|)
9. Bit-wise XOR (^)
10. Bit-wise XNOR (~^)
11. Is equal (==)
12. Is greater than (>)
13. Is less than (<)< li>
14. Shift right (>>1)
15. Shift left (<<1)< li>

System's Commands:

- Register file write command. This command consists of 3 frames as follows:
    1. Command opcode (0xAA)

    2. Register file write address
    3. Register file write data
- Register file read command. This command consists of 2 frames as follows:
    1. Command opcode (0xBB)
    2. Register file read address
- ALU operation with operands command. The operands of the ALU are connected to the first two registers of the register file, so to execute this command: the operands are first written to the first two registers in the register file then the result is evaluated. This command consists of 4 frames as follows:
    1. Command opcode (0xCC)
    2. Operand A
    3. Operand B
    4. ALU function
- ALU operation without operands command. This command executes the ALU operation on the stored values in the first two registers in the register file directly. This command consists of 2 frames as follows:
    1. Command opcode (0xDD)
    2. ALU function

In all ALU commands, the UART transmitter sends two consecutive frames (becuase the size of the ALU result is double the size of the frame).

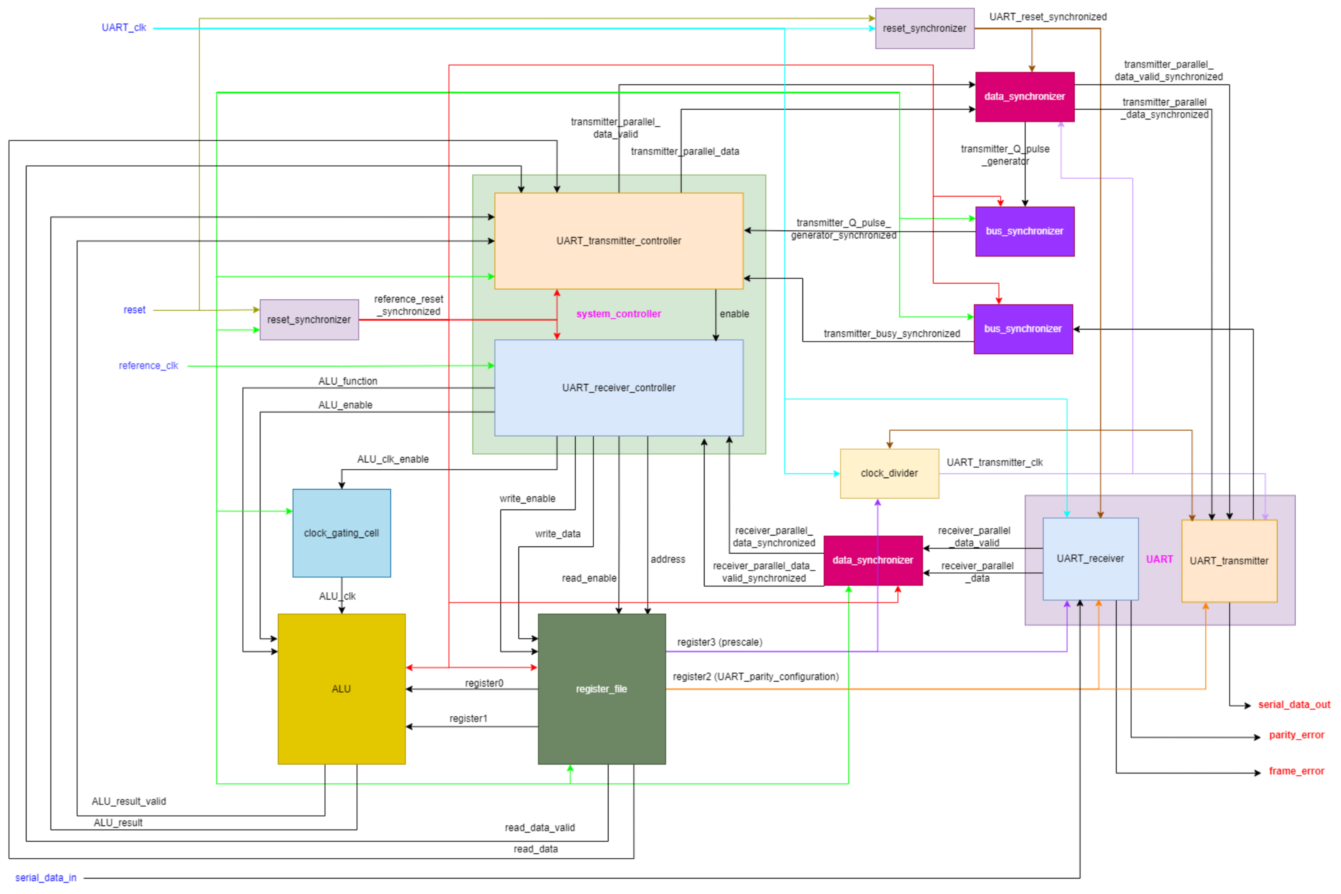
System's Configurations

- The parity configuration of UART (parity enable and parity type).
- The oversampling prescale (division ratio) of the UART receiver.

Note that the mentioned configurations are outputs from the register file (reference clock domain) and they are inputs to blocks that operates on UART clock (i.e. Metastability may occur becuase the source and destination domains are asynchronous to one another), however there is no synchronizers used to synchronize those signals because they are **Quasi-static signals** (they are effectively stable for long periods of time. Such domain crossings do not require synchronizers in the destination domain, because they are held long enough to be captured by even the slowest clock domains without the risk of metastability).

System Top Level Module

Block Diagram



Blue-colored signals are the system's input ports, while red-colored signals are the system's output ports.

Port Description

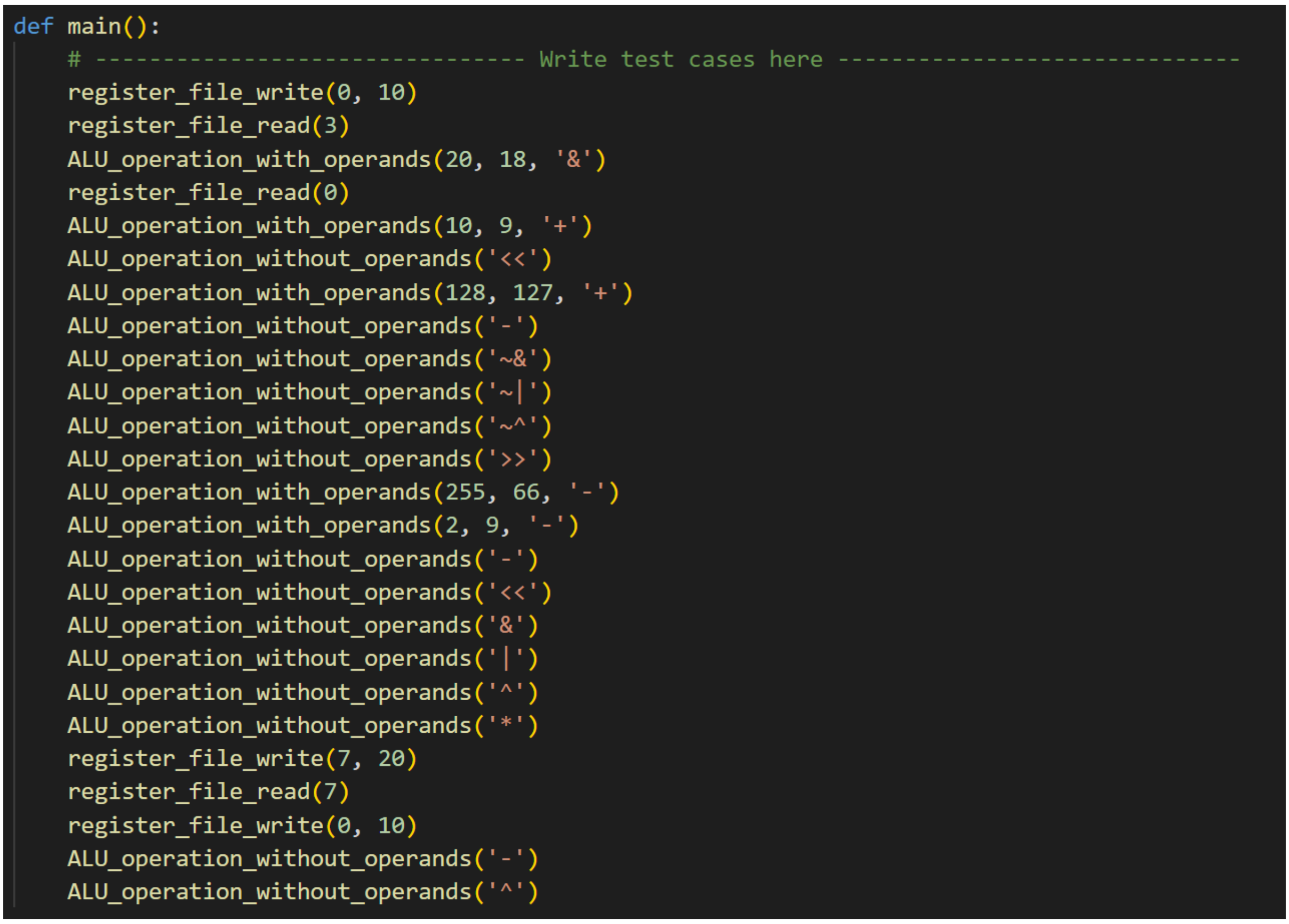
Port	Direction	Width	Description
reference_clk	input	1	The main clock of the system.
UART_clk	input	1	UART clock (the clock of the UART receiver).
reset	input	1	Unsynchronized global active low asynchronous reset.
serial_data_in	input	1	The data which is received serially by the UART receiver.
serial_data_out	output	1	The output of the UART transmitter (It is also the output of the mux that select between start, serial data, parity, or stop bits according to the state of the transmission).
parity_error	output	1	A signal to indicate that there is parity mismatch between the received parity bit and the calculated parity bit.
frame_error	output	1	A signal to indicate that the start bit or the stop bit was incorrect.

Functional Verification

The whole system is verified through an automated Python environment which does the following:

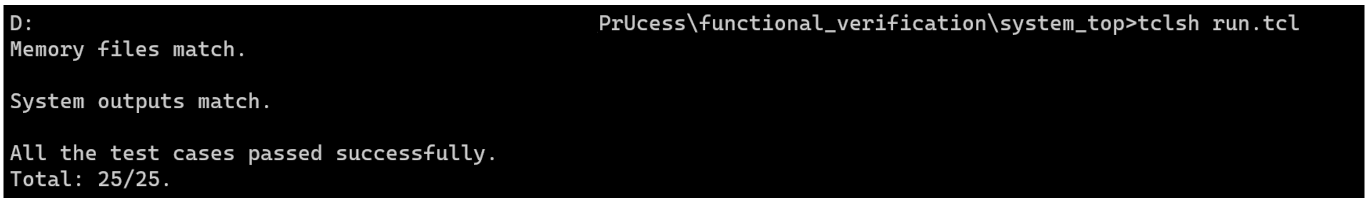
1. Generates the opcodes of all the given commands in an external file.
2. Generates all the expected results that should be transmitted serially through the UART transmitter in an external file.
3. Generates the memory file which corresponds to the final values that should be stored in the register file after the execution of all the commands.
4. Compares the results of the Verilog testbench (transmitted through UART transmitter) and the generated memory file with the expected results' file and expected memory file.
5. Reports any mismatch that occur in the testbench.
6. Reports the number of passed and failed testcases.

Sample test cases: Any arbitrary test case can be written in `functional_verification/system_top/test_cases_generator.py` as the following:



```python
def main():
    # -------------------------------- Write test cases here ------------------------------
    register_file_write(0, 10)
    register_file_read(3)
    ALU_operation_with_operands(20, 18, '&')
    register_file_read(0)
    ALU_operation_with_operands(10, 9, '+')
    ALU_operation_without_operands('<<')
    ALU_operation_with_operands(128, 127, '+')
    ALU_operation_without_operands('-')
    ALU_operation_without_operands('~&')
    ALU_operation_without_operands('~|')
    ALU_operation_without_operands('~^')
    ALU_operation_without_operands('>>')
    ALU_operation_with_operands(255, 66, '-')
    ALU_operation_with_operands(2, 9, '-')
    ALU_operation_without_operands('-')
    ALU_operation_without_operands('<<')
    ALU_operation_without_operands('&')
    ALU_operation_without_operands('|')
    ALU_operation_without_operands('^')
    ALU_operation_without_operands('*')
    register_file_write(7, 20)
    register_file_read(7)
    register_file_write(0, 10)
    ALU_operation_without_operands('-')
    ALU_operation_without_operands('^')
```

Then the script `functional_verification/system_top/run.tcl` is used to: run the Python script to generate the expected results, run the testbench to generate the actual results, and compare both the results to report any mismatch in the results.



```
D:                                             PrUcess\functional_verification\system_top>tclsh run.tcl
Memory files match.

System outputs match.

All the test cases passed successfully.
Total: 25/25.
```